Preprint
UCRL-JP-200044

Pursuing Scalability for *hypre*'s Conceptual Interfaces

Robert D. Falgout, Jim E. Jones and Ulrike Meier Yang

This article was submitted to
ACM Transactions on Mathematical Software

September 4, 2003

This research was supported under the auspices of the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

Pursuing Scalability for *hypre*'s Conceptual Interfaces

Robert D. Falgout   Jim E. Jones   Ulrike Meier Yang[*]

Abstract

The software library *hypre* provides high performance preconditioners and solvers for the solution of large, sparse linear systems on massively parallel computers as well as conceptual interfaces that allow users to access the library in the way they naturally think about their problems. These interfaces include a stencil-based structured interface (`Struct`); a semi-structured interface (`semiStruct`), which is appropriate for applications that are mostly structured, e.g. block structured grids, composite grids in structured adaptive mesh refinement applications, and overset grids; a finite element interface (`FEI`) for unstructured problems, as well as a conventional linear-algebraic interface (`IJ`). It is extremely important to provide an efficient, scalable implementation of these interfaces in order to support the scalable solvers of the library, especially when using tens of thousands of processors. This paper describes the data structures, parallel implementation and resulting performance of the `IJ`, `Struct` and `semiStruct` interfaces. It investigates their scalability, presents successes as well as pitfalls of some of the approaches and suggests ways of dealing with them.

1 Introduction

The software library *hypre* [6, 1] provides high performance preconditioners and solvers for the solution of large, sparse linear systems on massively parallel computers. Its development was motivated by the need to provide users with advanced scalable parallel solvers and preconditioners that efficiently solve computationally challenging applications of increasing proportions. Issues of robustness, ease of use, flexibility and interoperability have also been important.

One of its attractive features is the provision of conceptual interfaces that allow users to access the library in the way they naturally think about their problems. For example, application developers that use structured grids, typically think of their problems in terms of stencils and grids, whereas for an application that uses unstructured grids and finite elements it is more natural to access the preconditioners and solvers via elements and element stiffness matrices.

Conceptual interfaces also decrease the coding burden for users. The most common interface used in libraries today is a linear-algebraic one. This interface requires that the user compute the mapping of their discretization to row-column entries in a matrix. This code can be quite complex; for example, consider the problem of ordering the equations and unknowns on the composite grids used in structured AMR codes. The use of a conceptual interface merely requires the user to input the information that defines the problem to be solved, leaving the forming of the actual linear system as a library implementation detail hidden from the user.

[*]Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, P.O. Box 808, L-561, Livermore, CA 94551.

Conceptual interfaces also provide access to a large array of powerful scalable linear solvers that need the extra information beyond just the matrix. For example, geometric multigrid (GMG) cannot be used through a linear-algebraic interface, since it is formulated in terms of grids. Similarly, in many cases, these interfaces allow the use of other data storage schemes that have less memory overhead and that provide for more efficient computational kernels.

The conceptual interfaces currently include a stencil-based structured interface (`Struct`); a semi-structured interface (`semiStruct`), which is appropriate for applications that are mostly structured, e.g. block structured grids, composite grids in structured mesh refinement applications, and overset grids; a finite-element interface (`FEI`) for unstructured problems; and a traditional linear-algebraic interface (`IJ`). A detailed discussion of the design and use of these interfaces can be found in [7].

Clearly, it is of utmost importance to provide an efficient, scalable implementation of these interfaces in order to support the scalable solvers of the library, especially when using tens of thousands of processors.

The primary focus of this paper is on the implementation and performance of the conceptual interfaces in *hypre*, particularly the `IJ`, the `Struct`, and the `semiStruct` interface. The `FEI` interface was implemented elsewhere [5] and is therefore not discussed in this paper.

In Section 2, we state our goals and define some terms that will be used throughout the paper. In the following sections, the data structures, parallel implementation and resulting performance are discussed for the `IJ` (Section 3), the `Struct` (Section 4) and the `semiStruct` (Section 5) interface. The complexities of the algorithms that set up the data structures and communication packages are analyzed.

2 Goals and Definitions

It is important to define our goals regarding scalability. There are two basic considerations that need to be made here: computations and storage. For scalable computations, our goal is to construct algorithms that depend on the data logarithmically or better. So, if p is the number of processors, we want our algorithms to require no more than $O(\log p)$ parallel operations. For storage costs, we allow no more than $O(p)$ memory units. However, thinking of parallel computers with 100,000 processors or more, our ultimate goal is to reduce storage costs to $O(\log p)$, and we will discuss approaches on how to achieve this for each interface, starting with the `IJ` interface (since it is the simplest), followed by the `Struct` interface, and finally the `semiStruct` interface, which combines both interfaces.

The paper is structured such that for each of the interfaces under consideration we first describe the data structures. The data structures are very different for each interface and strongly depend on the particular conception that motivated each interface. Then communication issues are discussed, particularly how communication is currently implemented and how it can be improved in some cases.

Below we define a few terms that will be used in the remainder of the paper and are common to all interfaces. It is assumed that at the beginning of the setup phase the user gives only local information to the processor. Consequently, this requires that each processor needs to somehow obtain neighborhood information. This is achieved by the creation of a communication package on each processor. A *communication package* is a data structure that contains the neighborhood information. Each package needs to contain the IDs of the neighbor processors. There are two types of neighbors. We will refer to neighbors from which the processor needs to receive data as *receive processors*. Those neighbors to which data need to be sent will be called *send processors*. Often a receive processor is also a send processor and vice versa. In a symmetric problem (here the definition of symmetry depends on the interface and data structure chosen) the set of receive processors is identical to the set of send processors, but in general it is important to distinguish between both types. A communication package needs to also contain information on how to encode or decode data that needs to be sent or received. The format of this information depends on the individual interface and the underlying data structures.

In the following sections the data structures and communication packages for the `IJ`, the `Struct`, and the `semiStruct` interface are described, as well as issues particular to each interface. The algorithms that are necessary to create the communication package and other needed information are analyzed with regard to parallel computation units and memory usage.

3 The Linear-Algebraic Interface (`IJ`)

The `IJ` interface is the traditional linear-algebraic interface. Here, the user defines the right hand side and the matrix in the general linear-algebraic sense, i.e. in terms of row and column indices. This interface provides access only to the most general data structures and solvers (such as *BoomerAMG* [8], Euclid [9] and ParaSails [4]) and as such should only be used when none of the grid-based interfaces is applicable.

3.1 Data Structure

As with the other interfaces in *hypre*, the `IJ` interface expects to get the data in distributed form. Matrices are assumed to be distributed across p processors by contiguous blocks of rows. That is, the matrix must be blocked as follows:

$$\begin{pmatrix} A_1 \\ A_2 \\ \vdots \\ A_p \end{pmatrix}, \tag{1}$$

where each submatrix A_k is "owned" by a single processor k, $k = 1, \ldots, p$. A_k contains the rows $r_k, r_k + 1, ..., r_{k+1} - 1$.

We will make the additional assumptions, on which we base the analysis of the algorithms. Define N the global number of rows of A. $[r_1, \ldots, r_p, r_{p+1}]$ be the global partitioning of A, where $r_1 = 1$ and $r_{p+1} = N + 1$. Let us assume that N_b is the maximal number of rows per processor, i.e. $N_b = \max_{1 \le k \le p}(r_{k+1} - r_k)$, and that N_b is independent of p. This implies that $N \le pN_b$ and that N is increasing with increasing p. Often a balanced partitioning, i.e. $N_b \approx N/p$, is chosen in order to achieve good load balancing. Let us also assume that each processor has at most q neighbor processors and that each row of A has at most m coefficients, where q and m are independent of p. This is the case for many applications, e.g. for a 2-d 5-point Laplace operator $m = 5$ and $q = 4$ and for a 3-d 7-point Laplace operator $m = 7$ and $q = 6$.

To create an `IJ` matrix, the user needs specify the row extents, r_k and $r_{k+1} - 1$, on processor k. Next, the object type needs to be set, which determines the underlying data structure. Currently only one data structure, the ParCSR matrix data structure, is available.

Before we describe the ParCSR matrix, we give a brief definition of a CSR matrix, which is based on the sequential compressed sparse row (CSR) data structure. A *CSR matrix* consists of the integer NUM_ROWS, which defines the number of rows of the matrix, and the integer arrays ROW_START, COL_INDEX and DATA. DATA is a real array of the size of the number of non-zeros of the matrix and contains

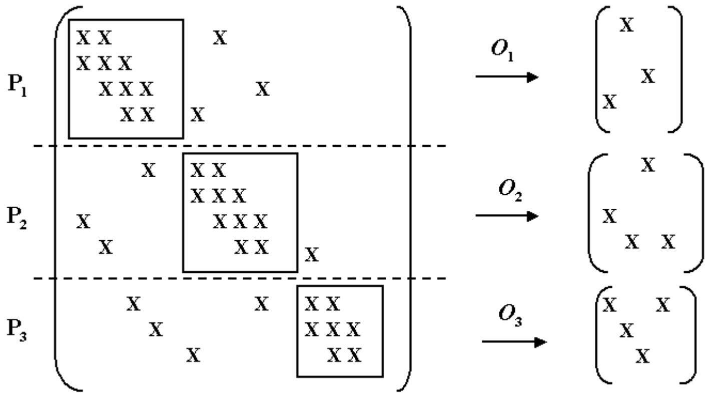

Figure 1: An example of a ParCSR matrix, distributed across 3 processors. Matrices with local coefficients, D_1, D_2 and D_3, are shown as boxes within each processor. The remaining coefficients are compressed into the matrices O_1, O_2 and O_3.

the non-zero coefficients of the matrix. Coefficients that belong to the same row need to be grouped together, although they need not be ordered within the row. Rows, however, need to be in order, i.e. row $k+1$ must follow row k immediately. COL_INDEX is an integer array of the same length and its k-th element contains the column index of the k-th element of DATA (COL_INDEX). ROW_START is an integer array of length NUM_ROW+1, and its i-th element points to the location of the first non-zero element of the i-th row of the matrix within array DATA.

A *ParCSR matrix* consists of p parts A_k, $k = 1, \dots, p$ (see (1)), where A_k is stored locally on processor k. Each A_k is split into two matrices D_k and O_k. D_k is a square matrix of order $n_k \times n_k$, where $n_k = r_{k+1} - r_k$ is the number of rows residing on processor k. D_k contains all coefficients a^k_{ij}, with $r_k \le i, j \le r_{k+1} - 1$, i.e. column indices pointing to rows stored locally. The second matrix O_k contains those coefficients of A_k, whose column indices j point to rows that are stored on other processors with $j < r_k$ or $j \ge r_{k+1}$. Both matrices are stored in CSR format. Whereas D_k is a CSR matrix in the usual sense, in O_k, which in general is extremely sparse with many zero columns and rows, all non-zero columns are renumbered for greater efficiency. Thus, one needs to generate an array of length n_{O_k} that defines the mapping of local to global column indices, where n_{O_k} is the number of non-zero columns of O_k. We denote this array as COL_MAP_O_k.

An example of an 11×11 matrix that illustrates this data structure is given in Figure 1. The matrix is distributed across 3 processors, with 4 rows on Processor 1 and Processor 2, and 3 rows on Processor 3. The 4×4 matrices D_1 and D_2 and the 3×3 matrix D_3 are illustrated as boxes. The remaining coefficients are compressed into the 4×3 matrix O_1 (with COL_MAP_O_1 = (5,6,8)), the 4×4 matrix O_2 (with COL_MAP_O_2 = (1,2,4,9)) and the 3×4 matrix O_3 (with COL_MAP_O_3 = (3,4,5,8)). Since often O_p is extremely sparse, efficiency can be further increased by introducing a row mapping that compresses zero rows by renumbering the non-zero rows.

COL_MAP_O_k can be generated by sorting all the column indices contained in O_k via an efficient sorting algorithm such as a quicksort and compress the resulting array so that each index appears only once. Quicksort applied to an array of size n has the computational complexity of order $O(n \log n)$ with a memory usage of $O(n)$. Due to the assumptions, the number of elements in O_k is limited by mN_b and independent of p. Often, the number of elements in O_k is much smaller than mN_b. The final array COL_MAP_O_k has at most as many, but often less elements than O_k, and its size is independent of p. It also has the additional advantage of being ordered, which allows the use of more efficient search algorithms, if one needs to find the local number of a global column index.

3.2 Generating the Communication Package

The communication package is based on the concept of what is needed for a matrix-vector multiplication. Let us consider the parallel multiplication of a matrix A with a vector $\mathbf{x}$. Processor k owns rows r_k through $r_{k+1}-1$ as well as the corresponding chunk of $\mathbf{x}$, $\mathbf{x}_k = (x_{r_k}, \ldots, x_{r_{k+1}-1})^T$. In order to multiply A with $\mathbf{x}$, Processor k needs to perform the operation $A_k\mathbf{x} = D_k\mathbf{x}_k + O_k\tilde{\mathbf{x}}_k$, where $\tilde{\mathbf{x}}_k = (x_{col\_map\_O_k(1)}, \ldots, x_{col\_map\_O_k(n_{O_k})})^T$. While the multiplication of D_k and $\mathbf{x}_k$ can be performed without any communication, the elements of $\tilde{\mathbf{x}}_k$ are owned by the receive processors of k. Another necessary piece of information is the amount of data to be received by each processor. In general processor k owns elements of $\mathbf{x}$ that are needed by other processors. Consequently processor k needs to know the indices of the elements that need to be sent to each of its send processors.

In summary, the communication package on processor k consists of the following information:

- the IDs of the receive processors
- the size of data to be received by each processor
- the IDs of the send processors
- the indices of the elements that need to be sent to each send processor

The algorithms used to obtain this information are described and analyzed in the remainder of the section, using the assumptions stated in the previous section.

Recall that each processor by design has initially only local information available, i.e. its own range and the rows of the matrix that it owns. We decided that each processor should have the complete partitioning information. This is not necessary, but simplifies the implementation considerably. It has the disadvantage that it requires a memory usage of order $O(p)$, since the length of the partitioning array is $p+1$, which can be a problem for very large p. In Section 3.4, we will suggest an algorithm that decreases this order.

In order to find the global partitioning, each processor needs to send its local range to all the other processors. This can be done using MPI_ALLGATHER, which if implemented efficiently (using a tree structure), will take $O(\log p)$ operations. Once the global partitioning is known, each processor has enough information to determine its receive processors as well as the amount of data to be received.

Once the global partitioning has been generated, it is easy to determine the receive processors by performing a search for each element j of COL_MAP_O_k in the global partitioning of A. If $r_i \leq j < r_{i+1}$, row j or element j is owned by processor i. If we assume an arbitrary matrix, j could be on any processor other than processor k. Since the partitioning is ordered, one can use binary search, which will find this element in at most $O(\log p)$ number of operations. However it is possible to do this in a more efficient way. If we assume a balanced partitioning, a good initial guess would be $[(j-1)p/N]+1$, which might be the ID of the wanted processor or is close to it. So even a sequential search following the guess should lead to the wanted ID within a few steps, yielding an algorithm with $O(1)$ number of operations. If we have determined the receive processor for j, there is a high probability that the next element l of $\text{COL\_MAP\_}O_k$ is owned by the same receive processor, say processor s. So for the following elements of $\text{COL\_MAP\_}O_k$, we check if it is owned by processor s first. Only if this is not the case, we conduct another search. If it is not owned by processor s, it must be owned by processor r where $r > s$, since both $\text{COL\_MAP\_}O_k$ as well as the partitioning are ordered. This restricts the search even further. This procedure is repeated until all receive processors have been determined. At the same time, one can determine the number of receive processors and the amount of data to be received by each receive processor. The search is performed at most q times, since this is the maximal number of neighbors according to the assumption. Thus the complete algorithm takes at most $O(\log p)$ operations, but possibly only $O(1)$ operations, if the partitioning is balanced.

If matrix A has a symmetric structure, the receive processors are also send processors, and no further communication is necessary. However, this is different in the non-symmetric case. In the current implementation, the IDs of the receive processors and the amount of data to be obtained from each receive processor are communicated to all processors. This is done via a MPI_Allgatherv, which requires $O(\log p)$ computations and $O(p)$ memory. Once this information has been received, each processor needs to search for its own ID in the information buffer. Since this buffer is not ordered, this requires at most $O(p)$ number of operations. For a moderate number of processors, even up to 1000, the number of operations is fairly insignificant compared to the size of mN_b, however it can become a potentially high cost if we consider 100,000 processors. An algorithm that determines the send processor in at most $O(\log p)$ number of operations is described in Section 3.3.

When each processor knows its receive and send processors, the remaining necessary information can be sent directly to the receive processors, which know now its send processors as well the amount of data to be received. Since the number of neighbors and the amount of data is independent of p, the computational complexity and the storage requirement is of order $O(1)$.

3.3 Determing the Send Processors Using a Binary Tree

In this section, we discuss a possible approach to determine send processors that eliminates the $O(p)$ computational complexity in the current implementation. The approach requires the use of the MPI_Iprobe function, which checks for unexpected messages sent by an unknown processor. Since its implementation may be inefficient, the use of MPI_Iprobe is generally not recommended; it is better to send a message to a processor that expects this message and has posted a corresponding receive statement. However, in the case of determining send processors, it is not possible for a processor to know which processors it will communicate with.

To illustrate the algorithm, assume that processor k determines processor r is one of its receive processors, thus making k one of r's send processors (a fact r cannot determine from its local information.) Processor k sends r a message informing it of this, which processor r will receive as it probes continually for incoming messages. However, processor r may have other send processors, it cannot determine how many such unexpected messages to receive. The question is: when can the processor stop probing, or how does it know that the last message has been received?

A binary tree structure can be used to determine when all messages have been received. First, it is necessary that each processor after receiving a message signals to the sending processor that it has received the message. Then the sending processor knows when all its messages have been received. Let us assume a binary tree structure as illustrated in Figure 2. Each processor has a parent, and each parent has up to two children, one of its children being itself. When a processor knows that

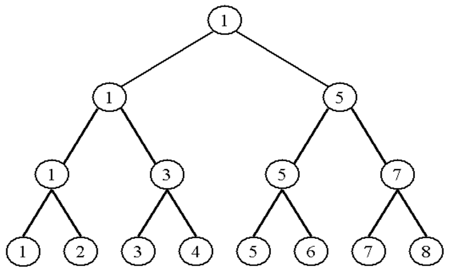

Figure 2: Binary tree structure with 8 processors

all its messages have been received, it sends a message to its parent. As soon as the parent has received a confirmation from all its children that they have received their messages, it sends a message to its parent, etc. When processor 1 finally receives the message, all messages have been received. Because of the tree structure, this takes $O(\log p)$ operations. Using the same tree structure in reverse order, processor 1 can now notify its children and stop probing, its children can in turn notify their children and stop probing, and so on. Note that in the case where a processor is its own parent it does not need to send a message to itself. After $O(\log p)$ operations, all processors have been notified and have stopped probing for incoming messages.

3.4 Using an Assumed Partitioning

One drawback of the current implementation of the `IJ` interface is the $O(p)$ memory use, caused by storing the global partitioning on each processor. For the most part, this information is not necessary, it is only used to determine the receive processors. One can avoid this by using the following approach involving a fixed *assumed partitioning*. This partitioning is defined by a function that is known by all processors and involves $O(1)$ operations (for a balanced partitioning this function could be $f(j) = [(j-1)p/N] + 1$.) In general, this function does not describe the actual partitioning, although one would hope that it is not too different.

This approach consists of three steps. First each processor k determines where its *assumed rows*, rows its responsible for in the assumed partition, are stored in the actual partition. Then it can compute its *assumed receive processors*, those processors responsible in the assumed partitioning for the column indices given in $\text{COL\_MAP\_}O_k$, and send this information to them. At the same time it receives such information from other processors. It can now compare this information with the actual ownership information that it obtained in the first step and return corrections or confirmations to the processors it received the information from. In the final step, the assumed receive processor information is updated by the actual receive processor information.

The first step can be implemented as follows. First, each processor can check its own actual range against this function and determine if it coincides with the assumed partitioning. If this is not the case, it needs to determine which processors are responsible for (in the assumed partitioning) the rows it owns in the actual partition and send this information to these processors. If it does not own rows that are within its range according to the assumed partitioning, it needs to post receives for an arbitrary processor - using MPI_ANY_SOURCE. Since its rows might be distributed across several processors, this requires a loop, in which messages are received until the processor knows where all its assumed rows are stored in the actual partition.

In the second step, each processor needs to first determine its assumed receive processors using the given function. This takes only $O(1)$ operations. Let us assume processor k determines that its assumed receive processors are processor r and processor s. Processor k sends the indices of the elements it needs to the processor responsible for them in the assumed partition. When r and s receive these messages by processor k, they check whether they actually own the elements or rows that k needs. If they own them, they send a message confirming this information to processor k. If they do not own them or only own a few of them, they know on which processor(s) the other rows or elements are located. Consequently, they can send this information to processor k. Since each processor has at most q neighbors and q is independent of p, the number of sends and receives is independent of p. However, since a processor does not know how many messages to expect, it is necessary to use a binary tree approach as described in Section 3.3 to inform a processor when to stop probing. This approach takes $O(\log p)$ number of operations, i.e. possibly more than the "lucky guess" approach, which might take only $O(1)$ operations, but memory usage is independent of p.

Finally, after each processor has stopped probing, it has received all the information it needs to update the assumed receive processor information by the actual receive processor information. In the best case, i.e the assumed receive processors are the actual receive processors and own the needed information, no change is necessary. In the worst case, everything needs to be updated. The number of computations and memory use in this step are independent of p.

3.5 Scalability Study

Figure (3) shows timings that were achieved by setting up increasingly larger matrices across larger number of processors. Two test cases are considered. The first one is a matrix derived by finite differences from a 3-dimensional Laplace operator with a 7-point stencil, the second matrix has a 27-point stencil on a cube. Each matrix has 64,000 (40x40x40) rows per processor and is set up by inserting a row at a time on each processor. The test runs were done using n = 1, 8, 27, 64, 125, 216, 343, 512, 729, 900 processors of the ASCI White computer. In the first test problem, each processor has at most 6 neighbors, in the second case at most 26 neighbors. The results show that the setup (which includes the generation of the communication package) is very scalable after an initial performance degradation. Obviously in the case of 1 processor, there is no communication and setup is done very fast. The increase in time is expected for the 8 processor case. Time increases further from the 8 (2x2x2) to the 27 (3x3x3) processor case. This is caused by the fact that for the first test problem each processor has only 3 neighbors, when using 8 processors, vs. at most 6 neighbors, using 27 and more processors. For the second problem, this difference is even more significant with 7 neighbors per processor, when using 8 processors, vs. up to 26 neighbors in the later test runs.

4 The Structured-Grid Interface (`Struct`)

The `Struct` interface is appropriate for scalar applications on structured grids with a fixed stencil pattern of non-zeros at each grid point. It provides access to *hypre*'s most efficient scalable solvers for scalar structured-grid applications, the geometric multigrid methods SMG [3] and PFMG [2]. The user defines the *grid* and the *stencil*; the matrix and right-hand-side vector are then defined in terms of the grid and the stencil.

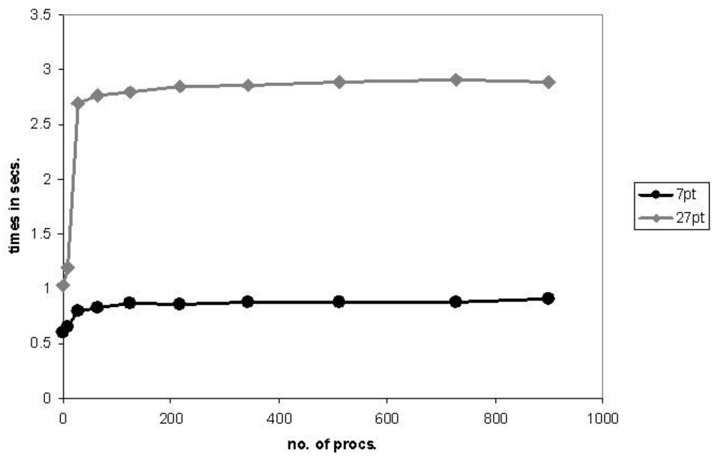

Figure 3: Matrix setup times for the IJ interface with increasing number of processors

The grid is described via a global d-dimensional *index space*, i.e. via integer singles in 1D, tuples in 2D, or triples in 3D (the integers may have any value, positive or negative). The global indices are used to discern how data is related spatially, and how it is distributed across the parallel machine. The basic component of the grid is a *box*: a collection of abstract cell-centered indices in index space, described by its "lower" and "upper" corner indices (see Figure 4). The scalar grid data is always logically associated with cell centers. Each process describes the portion of the grid that it "owns", one box at a time. Note that it is assumed that the data has already been distributed, and that it is handed to the library in this distributed form.

The stencil is described by an array of integer indices, each representing a relative offset (in index space) from some grid-point on the grid. For example, the geometry of the standard 5-pt stencil can be represented in the following way:

$$\begin{bmatrix} & (0,1) & \\ (-1,0) & (0,0) & (1,0) \\ & (0,-1) & \end{bmatrix}. \tag{2}$$

After the grid and stencil are defined, the matrix coefficients are passed as an array of doubles with each processor setting matrix values for the boxes it owns.

4.1 Data Structure

The underlying matrix data structure, *Struct matrix*, contains the following.

- *Struct grid*: describes the boxes owned by the processor (local boxes) as well as information about other nearby boxes. Note that a box is stored by its "lower" and "upper" indices, called the box's *extents*.
- *Struct stencil*: an array of indices defining the coupling pattern in the matrix.
- *data*: an array of doubles defining the coupling coefficients in the matrix.

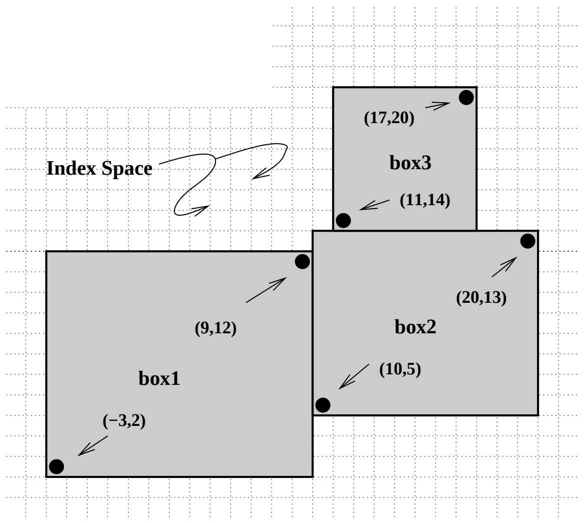

Figure 4: A box is a collection of abstract cell-centered indices, described by its minimum and maximum indices. Here, three boxes are illustrated.

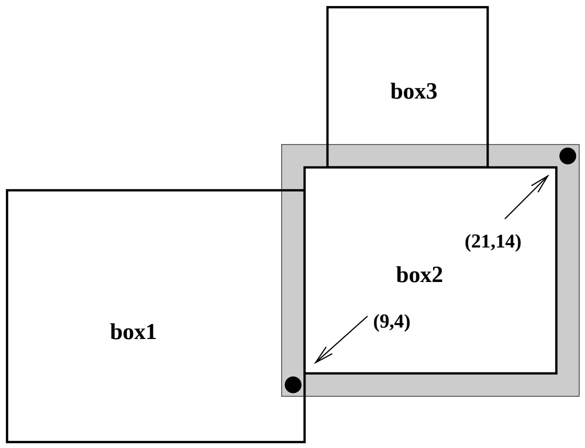

Figure 5: For parallel computing, additional storage is allocated for cells nearby a box (ghost cells). Here, the ghost cells for BOX2are illustrated.

The corresponding vector data structure is similar except is has no stencil and the data array defines the vector values. In both the vector and matrix the data array is stored so that all values associated with a given box are stored contiguously. To facilitate parallel implementation of a matrix-vector product, the vector data array includes space for values associated with a box somewhat larger than the actual box; typically including one boundary layer of cells or ghost cells (see Figure 5). Some of these ghost cells may be part of other boxes, owned by either the same or a different processor. Updating values in these ghost cells requires either copying data (if the neighbor box is owned by the same processor) or communicating data (if the neighbor box is owned by a different processor.) Determining these patterns for updating ghost cells is the major task in implementing the `Struct` interface in a scalable manor. In our implementation, there is much in common between the two cases (neighbor box on same or different processor) so in our discussion we will focus on the second case.

Assuming that the boxes are large, the additional storage of these ghost cells is fairly small as the boundary points also take only a small percentage of the total number of points. In some cases, the matrix data array will also contain space for ghost cells. In particular, our implementation allows reduced storage when the matrix is symmetric. For example, in the 5-pt stencil (Equation 2), the coefficients for the "west" coefficient may be explicitly stored and the "east" coefficient is defined by symmetry, i.e. the east coefficient at grid-point (i, j) is defined by the stored west coefficient at $(i + 1, j)$. This requires ghost cells for the matrix data; if the grid-point (i, j) is at the right-most boundary of a box its east coefficient is stored as the west coefficient in the ghost cell $(i + 1, j)$.

4.2 Generating the Neighborhood Information

Recall that in the interface, a given processor k is passed only information about the grid boxes that it owns. Determining how to update ghost cell values requires information about nearby boxes on other processors. This information is generated and stored when the Struct grid is assembled. Determining which processors own ghost cells is similar to the problem in the `IJ` interface of determining the receive processors. In the `IJ` case, this requires information about the global partitioning. In the `Struct` case, it requires information about the global grid.

The algorithm proceeds as follows. Here we let p denote the number of processors and b denote the total number of boxes in the grid (note $b \geq p$). First we accumulate information about the global grid by each processor sending the extents of its boxes to all other processors. As in the `IJ` case, this can be done using MPI_ALLGATHER with $O(\log p)$ operations. Memory usage is of order $O(b)$, since the global grid contains b boxes.

Once the global grid is known, each local box on processor k is compared to every box in the global grid. In this box-by-box comparison a *distance index* is computed which describes the spatial relationship in the index space of one box to another. If box a is shifted by the distance index, $d(a, b)$, it will define a region in index space that intersects box b. Further, this is the minimal shift producing a non-empty intersection. The *distance* between two boxes is defined by the minimum absolute value of the distance index components. As an example, BOX1 and BOX3 in Figure 4 are distance 2 apart with $d(\text{BOX1},\text{BOX3}) = (2, 2)$ and $d(\text{BOX3},\text{BOX1}) = (-2, -2)$ This comparison of each local box to every global box involves $O(b)$ computations.

Once the comparison is done, all global boxes within a specified distance (typically 2) from a local box are stored as part of a *neighborhood* data structure on processor k. Boxes not in this neighborhood can be deleted from processor k's description of the global grid. The storage requirement for the neighborhood is independent of p. The neighborhood data structure contains information about the nearby boxes: their extents, the processor owning it, and a unique ID (each box in the grid has an associated unique ID number.) In addition, it contains a linked list structure that quickly gives information about which neighborhood boxes intersect a given local box when the local box is shifted by a particular index.

The current neighborhood structure does give fast access to information about nearby cells owned by other processors, but there are potential drawbacks. The storage requirement is $O(b)$ as the global grid is initially gathered to all processors and the box-by-box comparison to determine neighbors involves $O(b)$ operations. One possible approach to eliminate these drawbacks would be similar to the assumed partitioning approach described at the end of Section 3.4. The idea is to have a function describing an assumed partitioning of the index space to processors and have this function available to all processors. Unlike the one-dimensional `IJ` partitioning, this partition would be d-dimensional. A processor would be able to determine its neighbors in the assumed partition in $O(1)$ computations and storage.

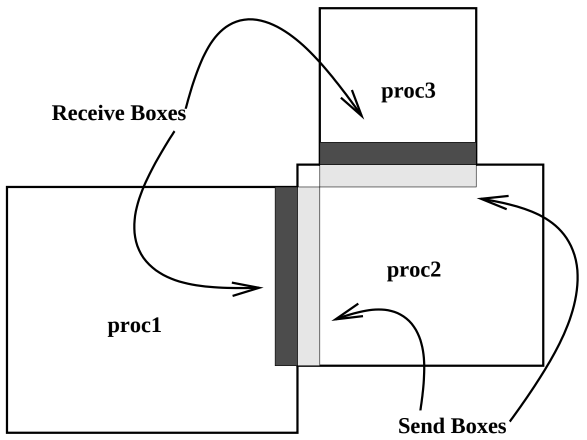

Figure 6: The communication package for processor 2 contains send boxes (values owned by processor 2 needed by other processors) and receive boxes (values owned by other processors need by processor 2.)

A multi-phase communication procedure like that previously described for the `IJ` case could be used to determine the actual neighbors with $O(\log p)$ complexity.

4.3 Generating the Communication Package

In this section, we describe the generation of a communication package which includes information needed for updating ghost cell values. As mentioned, ghost values in the vector are needed to perform a matrix-vector product and ghost values in the matrix are needed when symmetric storage is used. Here we concentrate on the matrix-vector product.

To perform the matrix-vector product, processor k must have up-to-date values in all ghost cells that will be "touched" when applying the matrix stencil at the cells owned by processor k. Determining these needed ghost cells is done by taking each box owned by the processor, shifting it by each stencil entry and finding the intersection of the shifted box with boxes in the neighborhood data structure. Because of the linked list structure, the shifted box is intersected not with all neighbor boxes, but only those that can produce a non-zero intersection. As an example, consider the same layout of boxes as before with each box on a different processor (see Figure 6). If the matrix has the 5-pt stencil (Equation 2), then shifting BOX2 by the "north" stencil entry and intersecting this with BOX3 produces one of the dark shaded regions labeled as a receive box. Using this procedure, a list of receive boxes and corresponding owner processors is generated.

The procedure for determining the cells owned by processor k that are needed to update ghost cells on other processors is similar. Here we need to shift the neighbor boxes by each stencil entry and find the intersection with the local boxes. In Figure 6, shifting BOX1 by the "east" coefficient and intersecting this with BOX2 produces one of the light shaded regions labeled as a send box. The linked list structure again limits the computations needed by considering only boxes that can produce non-zero intersections. This procedure produces a list of send boxes and corresponding list of processors needing these values.

4.4 Scalability Study

Figure (7) shows timings that were achieved by setting up increasingly larger matrices across larger number of processors. The matrix is derived by finite differences

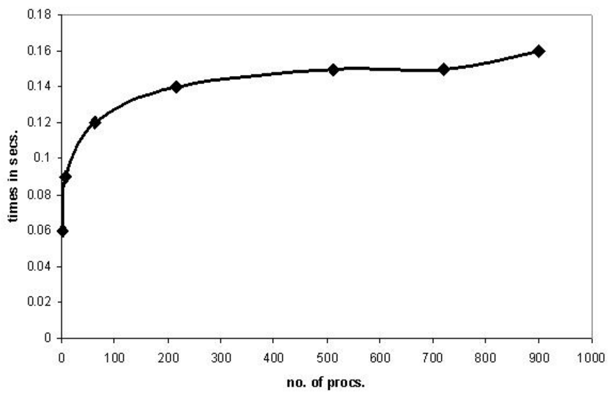

Figure 7: Matrix setup times for the Struct interface with increasing number of processors

from a 3-dimensional Laplace operator with a 7-point stencil. Each processor owns a single box of size $40 \times 40 \times 40$. The test runs were done using 1, 8, 64, 216, 512, 720 and 900 processors of the ASCI White computer. This setup includes the generation of the neighborhood structure. Since symmetric storage was used, it also includes the creation of a communication package to update ghost values in the matrix and the actual resulting MPI communication. As was the case for the `IJ` interface, the results show that the setup is very scalable after the initial hit for communications. For this number of processors, we do not see much effect from either the $O(\log p)$ operations in gathering information about the global grid using MPI_Allgather, or the more significant $O(b)$ operations needed in the box-by-box comparison to determine neighbors. Note that these times to build the matrices are quite small; much smaller than the times needed to solve the linear system. Also the time difference between the 720 and 900 processor runs is at the level of the resolution of the timer we used - 0.01 seconds.

5 The Semi-Structured-Grid Interface (`semiStruct`)

The `semiStruct` interface is appropriate for applications with grids that are mostly—but not entirely—structured, e.g. block-structured grids (see Fig. 8), composite grids in structured AMR (adaptive mesh refinement) applications, and overset grids. In addition, it supports more general PDEs than the `Struct` interface by allowing multiple variables (system PDEs) and multiple variable types (e.g. cell-centered, face-centered, etc.). The interface provides access to data structures and linear solvers in *hypre* that are designed for semi-structured grid problems, but also to the most general data structures and solvers.

The `semiStruct` grid is composed out of a number of structured grid *parts* each with its own index space, where the physical inter-relationship between the parts is arbitrary. Each part is constructed out of two basic components: boxes (see Section 4) and *variables*. Variables represent the actual unknown quantities in the grid, and are associated with the box indices in a variety of ways, depending on their types. In *hypre*, variables may be cell-centered, node-centered, face-centered,

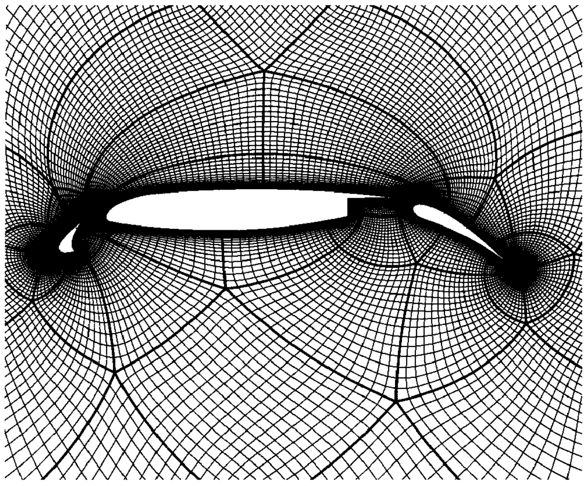

Figure 8: An example block-structured grid, distributed across many processors.

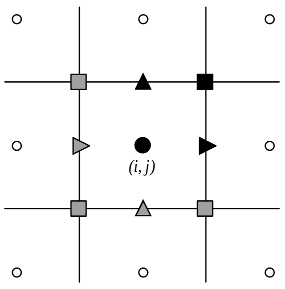

Figure 9: Grid variables in *hypre* are referenced by the abstract cell-centered index to the left and down in 2D (and analogously in 3D). So, in the figure, index (i, j) is used to reference the variables in black. The variables in gray, although contained in the pictured cell, are not referenced by the (i, j) index.

or edge-centered. Face-centered variables are split into x-face, y-face, and z-face, and edge-centered variables are split into x-edge, y-edge, and z-edge. The unknowns in the linear system are characterized by (PART, VAR, INDEX): a part number, a variable number, and an index identifying a particular cell on the part. See Figure 9 for an illustration in 2D.

The non-zero pattern of the matrix is described through a *graph*. The graph contains two types of couplings: *stencil* and *non-stencil* couplings. The stencil couplings describe a coupling pattern that is present throughout the grid and are described by stencils similar to the `Struct` case. The non-stencil couplings are specific couplings between particular unknowns, i.e. (PART1,VAR1,INDEX1) is coupled to (PART2,VAR2,INDEX2). The interface allows arbitrarily many non-stencil couplings and they may couple any unknowns, but the `semiStruct` interface is only appropriate when the majority of matrix non-zeroes are due to stencil couplings. In most cases, the stencil couplings describe the coupling within a part (*intra-part*) and the non-stencil couplings describe coupling between parts (*inter-part*). However, this is not always the case. Non-stencil entries may describe intra-part couplings that occur at only certain cells in the part and therefore do not belong to the stencil. Through the use of the `GridSetNeighborBox()` routine, stencil entries can describe inter-part couplings. This routine is used to describe how the index space on PART1 is related to the index space on PART2. This relationship allows stencil entries reaching "outside" of PART1 to touch variables on PART2. See [7] for a description of this usage in block-structured grids.

After the graph is defined, the matrix coefficients are passed as an array of doubles with each processor setting matrix values for the boxes it owns.

5.1 Data Structure

The `semiStruct` interface allows the user to choose from two underlying data structures for the matrix. One option is to use the ParCSR matrix data type discussed in Section 3.1. The second option is the *semiStruct matrix* data type which is based on a splitting of matrix non-zeros into structured and unstructured couplings $A = S + U$. The S matrix is stored as a collection of Struct matrices and the U matrix is stored as a ParCSR matrix. In our current implementation, the stencil couplings within variables of the same type are stored in S all other couplings are stored in U. If the user selects the ParCSR data type, then all couplings are stored in U (i.e. $S = 0$.)

Since the `semiStruct` interface can use both Struct and ParCSR matrices, the issues discussed in the previous two sections impact its scalability as well. The major new issue impacting scalability is the need to relate the the semi-structured description of unknowns and the global ordering of unknowns in the ParCSR matrix, i.e. the mapping $M(\text{PART},\text{VAR},\text{INDEX}) = \text{GLOBAL\_RANK}$. The implementation needs this mapping to set matrix entries in U.

5.2 Mapping to Global Ranks

The global ordering of unknowns is an issue internal to the `semiStruct` implementation; the user does not need and is not aware of this ordering. In our implementation the ordering is defined as follows.

```
GLOBAL_RANK= 0
loop over processors
  loop over variables
    loop over parts
      loop over boxes
        loop over grid indices in box
          $M$(PART,VAR,INDEX) = GLOBAL_RANK
          GLOBAL_RANK = GLOBAL_RANK+1
```

In our implementation of the semi-structured grid we include the concept of BoxMap to implement this mapping. There is a Box Map for each variable on each part; the purpose is to quickly compute the global rank corresponding to a particular index. To describe the BoxMap structure we refer to Figure 10. By cutting the index space in each direction by lines coinciding with boxes in the grid, the index space is divided into regions where each region is a either empty (not part of the grid) or is a subset of a box defining the grid. The data structure for the BoxMap corresponds to a d-dimensional table of BoxMapEntries. In three dimensions, BoxMapEntry$[i][j][k]$ contains information about the region bounded by cuts i and $i+1$ in the first coordinate direction, cuts j and $j+1$ in the second coordinate direction, cuts k and $k+1$ in the third coordinate direction. Among the information contained in BoxMapEntry is the first global rank (called *offset*) and the extents for the grid box which this region is a subset of. The global rank of any index in this region can be easily computed from this information.

The mapping $M(\text{PART},\text{VAR},\text{INDEX}) = \text{GLOBAL\_RANK}$ is computed by accessing the BoxMap corresponding to PART and VAR, searching in each coordinate direction

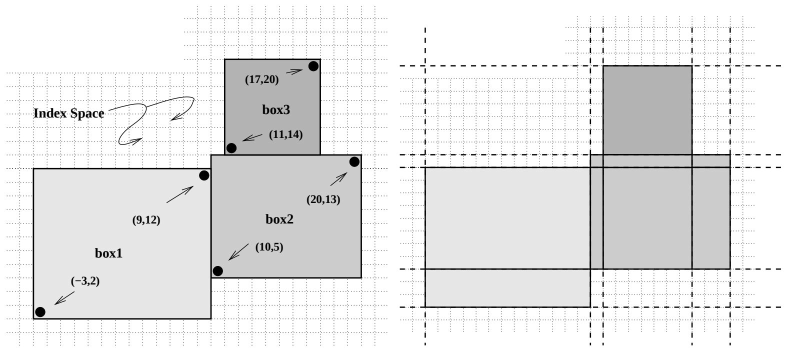

Figure 10: The BoxMap structure divides the index space into regions defined by cuts in each coordinate direction.

to determine which cuts index falls between, retrieving the offset and box extents from the appropriate BoxMapEntry, and computing global_rank from this retrieved information. This computation has $O(1)$ (independent of number of boxes and processors) complexity except for the searching step. The searching is done by a simple linear search so worst case complexity is $O(b)$ since the number of cuts is proportional to the number of boxes. However, we retain the current position in the BoxMap table, and in subsequent calls to the mapping function, we begin searching from this position. In most applications, subsequent calls will map indices nearby the previous index and the search has $O(1)$ complexity. Further optimization is accomplished by retrieving BoxMapEntries not for a single index but for an entire box of indices in the index space.

The BoxMap structure does allow quick mapping from the semi-structured description to the global ordering of the ParCSR matrix, but it does have drawbacks: storage and computational complexity of initial construction. Since we store the structure on all processors, the storage costs are $O(b)$ where b is the global number of boxes (again b is at least as large as p, the number of processors). Constructing the structure requires knowledge of all boxes (accomplished by the MPI_Allgather with $O(\log p)$ operations and $O(b)$ storage as in the `Struct` case), and then scanning the boxes to define the cuts in index space (requiring $O(b)$ operations and storage.) As in the `IJ` and `Struct` cases, it may be possible to use the notion of an assumed partitioning of the index space to remove these potential scalability issues.

5.3 Scalability Study

Figure (12) shows timings that were achieved by setting up increasingly larger matrices across larger number of processors. The matrix is again derived by finite differences from a 3-dimensional Laplace operator with a 7-point stencil. Three different descriptions or partitions of the grid were used (see Figure 11). In partition 1, the grid is defined as a single part and each processor owns a single box of size $40 \times 40 \times 40$. In partition 2, the grid is defined as two parts and each processor owns a box of size $40 \times 40 \times 20$ on each part. In partition 3, the grid is defined as two parts and each processor owns a box of size $40 \times 40 \times 40$ on one of the parts. Times are shown for using both the semiStruct and ParCSR data structures. The test runs were done using 1, 8, 64, 216, 512, 720 and 900 processors of the ASCI White computer. This setup includes the generation of the BoxMap structure and (in the ParCSR runs) the use of it to map to the global ordering. In the results, we first note that the three different partitions all perform similarly. The biggest effect is the choice of underlying data type with the semiStruct type being much faster than the ParCSR. The speed advantage of semiStruct data type is likely due to the ability to

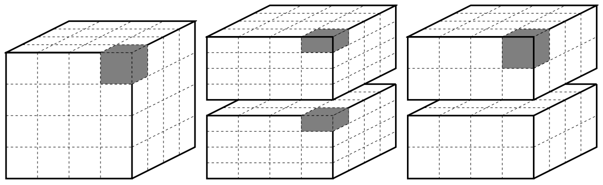

Figure 11: Layout of grids in numerical experiments: partition 1 (left) , partition 2 (middle), partition 3 (right).

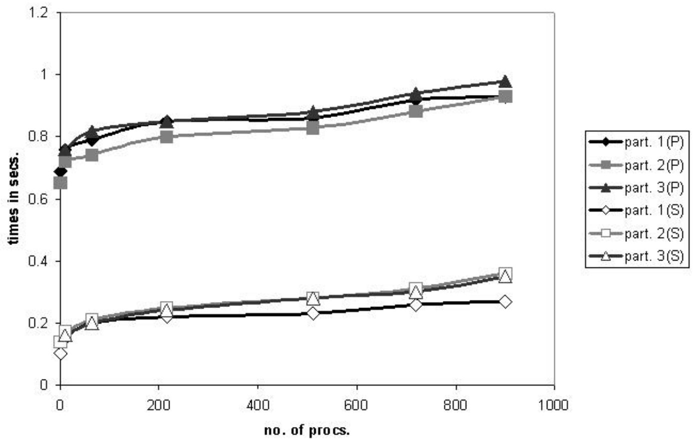

Figure 12: Matrix setup times for the semiStruct interface with increasing number of processors. There are two time shown for each partition: one with the semiStruct data type (S) and one with the ParCSR data type (P).

quickly set coefficients corresponding to a box of index space simultaneously in the underlying Struct matrix. In the results, we see somewhat worse scaling behavior than either `IJ` or `Struct`. This is to be expected. Since the **`semiStruct`** interface uses both Struct and ParCSR matrices and is built on top of the `IJ` and `Struct` interfaces, it inherits any of their potential scalability problems. In addition, there is the additional $O(b)$ computations associated with the construction of the BoxMap structure. Although these potential scaling problems exist, their effect is not that large for the number of processors in this study, and the interface scales reasonably well.

6 Conclusions and Future Work

The experiments show that for a moderate number of processors the various *hypre* interfaces are very scalable. The analysis shows that parts of the interface have some scalability issues that could be of concern when using 100,000 processors. However, we have suggested several scalable algorithms that deal with these issues, and that we plan to implement in the future. Future plans also include adding other data structures to the `IJ` interface. Additional data structures are desirable for various reasons, e.g. to be able to link to other packages, such as PETSc. Also, if additional matrix information is known, more efficient data structures are possible.

For example, if the matrix is symmetric, it would be advantageous to design a data structure that takes advantage of symmetry. Such an approach could lead to a significant decrease in memory usage. Another data structure could be based on blocks and thus make better use of the cache. Small blocks could naturally occur in matrices derived from systems of PDEs, and be processed more efficiently in an implementation of the nodal approach for systems AMG.

7 Additional Information

The *hypre* library can be downloaded by visiting the *hypre* home page at the URL `http://www.llnl.gov/CASC/hypre`. Although *hypre* is written in C, it can also be called from Fortran. Information on *hypre* and how to use it can be found in the users manual and the reference manual, which are also available at the same URL.

Acknowledgments

This paper would not have been possible without the many contributions of the *hypre* library developers: Edmond Chow, Andy Cleary, Van Henson, David Hysom, Mike Lambert, Jeff Painter, Charles Tong and Tom Treadway. This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

References

[1] *hypre*: high performance preconditioners. http://www.llnl.gov/CASC/hypre/

[2] Ashby, S., Falgout, R.: A parallel multigrid preconditioned conjugate gradient algorithm for groundwater flow simulations. Nuclear Science and Engineering **124** (1996) 145–159

[3] Brown, P., Falgout, R., Jones, J.: Semicoarsening multigrid on distributed memory machines. SIAM J. Sci. Comput. **21** (2000) 1823-1834

[4] Chow, E.: A priori sparsity patterns for parallel sparse approximate inverse preconditioners. SIAM J. Sci. Comput. **21** (2000) 1804-1822

[5] Clay, R., Mish, K., Otero, I., Taylor, L, Williams, A.: An annotated reference guide to the finite element interface (fei) specification: version 1.0. Sandia National Laboratories Report SAND99-8229, 1999.

[6] Falgout, R., Yang, U.M.: *hypre*: a library of high performance preconditioners. in Computational Science - ICCS 2002 Part III, Sloot, Tan, Dongarra, Hoekstra, eds., Lecture Notes in Computer Science **2331** (2002) 632-641, Springer. Also available as LLNL technical report UCRL-JC-146175.

[7] Falgout, R., Jones, J., Yang, U.M.: Conceptual interfaces in *hypre*. To appear in Future Generation Computer Systems. Also available as LLNL technical report UCRL-JC-148957.

[8] Henson, V. E., Yang, U. M.: *BoomerAMG*: a parallel algebraic multigrid solver and preconditioner. Applied Numerical Mathematics **41** (2002) 155-177. Also available as LLNL technical report UCRL-JC-133948 (2000).

[9] Hysom, D., Pothen, A.: A scalable parallel algorithm for incomplete factor preconditioning. SIAM J. Sci. Comput. **22** (2001) 2194–2215